Deploying Moodle on Azure

By Mario A. Di Vece

Abstract

Windows Azure supports running PHP applications via 2 capabilities: Native Code Execution, and FastCGI support. It also supports running a database engine in a worker role. Azure is designed to make applications highly reliable, highly available, and highly scalable. In this article we will dissect the migration of a popular, open source PHP application to Windows Azure. We will also cover the differences between running that same application with 2 different database engines: MySQL deployed in a worker role, and using SQL Azure. The overall objective of this article is to provide a practical example of modifying, configuring, and packaging an existing PHP web application in order to enable horizontal scalability, high availability and high performance using Windows Azure as a cloud computing platform.

Introduction

The goal of Windows Azure is to provide a practical platform on which applications are inherently highly available, highly reliable, and highly scalable. Windows Azure reduces the total cost of ownership by taking the required infrastructure to a large server cluster called “the cloud”. Windows Azure is designed to be elastic. This means that if an application requires more CPU, RAM, bandwidth, or disk space, instead of focusing on scaling vertically (i.e. adding more RAM or upgrading the CPU in the server that runs the application), you deploy copies – called instances – of that same application so that the load is evenly distributed, and a larger user base get incrementally supported.

If this is not the first article you read about deploying PHP applications on Windows Azure, you are probably wondering what is different about this one. Currently, most articles describing the process of deploying and running FastCGI applications on Windows Azure explain how to do such deployments in single-instance scenarios, and do not cover the architectural changes that are needed in order to tap into Azure’s full capabilities. In other words, it is easy to deploy single-instance web roles and worker roles. But when it comes to making applications highly available, scalable, reliable, we need to know what needs to change in the codebase in order get all of Azure’s benefits. Per the Windows Azure Compute SLA – or Service Level Agreement – Microsoft guarantees that when you deploy two or more role instances in different fault and upgrade domains your Internet facing roles will have external connectivity at least 99.95% of the time. For you, it means that you need to have at least 2 instances of a web application to benefit from the above statement.

Background

Before we get started, we must first understand the differences between running a PHP web application on a regular web server such as IIS or Apache, and running an application on Azure. In regular web servers, configuration settings usually point to paths on a local hard drive, file uploads are usually stored in a local directory structure, and session data usually resides in an OS-specific temporary path. In Windows Azure there is nothing preventing the application from creating temporary files, but there is also no such thing as a predictable hard drive letter mapping. All of the temporary files written by the application instance are not guaranteed to be persisted indefinitely. In other words, all files written to the "role's disk" are volatile. This is why we need to move away from writing files to local hard drives, and use Windows Azure Storage Services.

Windows Azure provides various means of persisting data in a scalable manner. We will focus on using Windows Azure Table Storage and Windows Azure Blob Storage services. These services are simple, REST-based services used to write data that is guaranteed to be persisted across time. If you are a seasoned PHP developer, you probably know that session data is usually stored in temporary files. In Windows Azure we will be using the Table Storage service instead of temporary files. The good news is that, there is a good library that you can effortlessly use to move session persistence to Azure's Table Storage service. Please refer to http://phpazure.codeplex.com/ to download the Windows Azure SDK for PHP. We will cover the specifics on configuring your PHP application to use Azure's Table Storage service for session persistence in a further section of this article. The bad news is that even though the Windows Azure SDK for PHP does have a custom stream wrapper, some advanced file operations required by a number of PHP applications are not compatible with the previously mentioned library's stream wrapper implementation. To complicate things, some PHP applications rely on multi-level directory structures which, Azure's Blob Storage service does not natively support. For these reasons, we had to modify the library in the SDK. This modified version provides:

- Support for OS-level file system functionality and
- Simulation of directory structures as blob name prefixes.

Another problem we face is the careful modification of the "php.ini" file for which we must keep in mind that paths cannot point to a local hard drive anymore. Configuration sections (such as the "extensions" section) which, heavily rely on path configuration entries, should now point to paths within the role's root path. Again, we will walk through the specifics in a further section of the article.

Finally, the last issue that we need addressed could be the largest, or the smallest of our problems, depending on the situation. Typically PHP applications fall within the following scenarios:

a. The web application uses MySQL as a database engine.
b. The web application is designed to use MSSQL as a database engine. The use of the older "php_mssql" PHP extension instead of the newer "php_sqlsrv" extension is common.
c. The application uses a database abstraction library such as ADOdb or PDO, which allows us to either write a new provider, or load a compatible extension.

For scenario a, we have 2 options: Load up MySQL in an Azure worker role, or migrate the data access code to use MSSQL with the "php_sqlsrv" driver. There are plenty of articles that show you how to run MySQL in a worker role. This approach is inconvenient, however, because you have to pay $0.12 an hour per worker role. That means you will end up paying $86.40 a month for a single instance of MySQL. And if you want to add failover support, you will need to create a second instance of MySQL in a worker role and configure it differently (i.e. as a slave). When comparing the work involved to deploy and maintain, and administer at least 2 MySQL instances in Azure worker roles, SQL Azure proves more convenient and far superior because you get a highly reliable, fast, and scalable 1GB database for only $9.99 a month. In short, it is strongly advised not to deploy MySQL in Windows Azure unless absolutely necessary, or until you have completely implemented SQL Server support for your application. Please refer to http://code.msdn.microsoft.com/winazuremysqlphp to get all the information you need about deploying MySQL on Windows Azure.

In scenario b, we encounter a typical situation in which an application has been developed with SQL Server as a database engine, but it relies on the "php_mssql" extension. The problem is that the "php_mssql" extension is not compatible with SQL Azure. You will need to re-implement SQL Server support on top of a different extension called "php_sqlsrv". You can download the latest binaries from http://www.microsoft.com/sqlserver/2005/en/us/PHP-Driver.aspx. There are numerous differences between the 2 extensions, especially when it comes to parameter-binding and query execution. Also, method names are different. In other words, even if you are using the same database engine, migrating application code to support the "php_sqlsrv" extension is not as straightforward as loading up a different PHP extension.

Scenario c is what we encounter in more mature, well-architected applications. A database-agnostic library provides access to a number of different database server technologies and it is only a matter of implementing support (if not already there) for the "php_sqlsrv" extension. For older libraries such as ADOdb this task might prove challenging and time-consuming. In this article, we will provide a practical example of modifying, configuring, and packaging an existing PHP web application in this scenario – Moodle – in order to enable horizontal scalability, high availability and high performance using Windows Azure as a cloud computing platform.

Deploying and Running MoodleAzure

In this guide, we will setup a local development environment by installing all the required extensions and downloading a modified version of Moodle 1.9.9 which is fully aware of both, Windows Azure and Sql Azure services. We will then package our solution, and deploy it on the cloud.

Setting up a Development Environment

1. The first step towards setting up a development environment is to ensure that you are running IIS. If you are unsure whether you are running IIS or not, go to: Start Menu > Control Panel > Administrative Tools > Services. Scroll all the way down and ensure the World Wide Web Publishing Service is listed, and started. If you cannot find this entry, go to Start Menu > Control Panel > Programs and Features > Turn Windows Features On or Off. Find and expand the

"Internet Information Services" node and make sure "Web Management Tools" and "World Wide Web Services" items are checked.

2. Install the latest version of PHP using Web Platform Installer. Go to http://www.microsoft.com/web/platform/phponwindows.aspx and install PHP 5 on IIS. We will use the downloaded files for the deployment we will setup later.
3. Install the WinCache extension. This extension caches PHP opcodes so that the script execution process only reads the script once and caches the opcodes. Subsequent requests are served using a "precompiled" version of the PHP script. You can use the Web PI to install it from the same page as the one we installed PHP from. http://www.microsoft.com/web/platform/phponwindows.aspx
4. Install the SQLSRV extension using Web PI. This extension provides Sql Azure compatibility with PHP. You can install this PHP extension by opening up the Web Platform Installer: Start Menu > All Programs > Microsoft Web Platform Installer. Click the Web Platform tab. Click the Customize link under the Database section. Finally check the "Microsoft Drivers for PHP for SQL Server 2.0. Finally, click on "Install"
5. At this point we have setup IIS, PHP, WinCache, and the Sql Server drivers for PHP. Now we will create the required set of folders in which we will place our solution files and prepare for deployment. Create a folder named "moodleazure" under "C:\projects" the resulting full path should be: "C:\projects\moodleazure". Create a second folder named "moodledata" under "C:\". The resulting full path should be: "C:\moodledata". This folder structure is only a suggestion. We will go over path configurations in the following section of this guide.

Packaging, Configuring and Deploying the Solution

Now that we have setup an appropriate development environment, we need to get the source code, add PHP binaries to our solution, configure our deployment, and run our application on the cloud.

1. Get the latest zip release of MoodleAzure source code from Codeplex (http://moodleazure.codeplex.com). You should be able to find it in under the "Downloads" section.
2. Extract the files to the folder "C:\projects\moodleazure\", which we created in step 5 of the previous section.
3. Go to your PHP installation folder. In x86 architectures, this should be "C:\Program Files\PHP". In x64 architectures it should be "C:\Program Files (x86)\PHP". Copy all of the files located inside the PHP installation folder EXCEPT FOR "php.ini" -- MoodleAzure source distributions comes preconfigured with a php.ini file --. Paste the copied files on to "C:\projects\moodleazure\MoodleWebRole\php"
4. Go to: Start Menu > All Programs > Microsoft Visual Studio 2010; right click on the Microsoft Visual Studio shortcut and select "Run as Administrator". Open up the Unosquare.MoodleAzure.sln file within Visual Studio.
5. Expand the MoodleWebRole project and configure MoodleAzure by providing the correct values for the settings listed in the table below.

File (line)	Setting	Value
config.php (7)	$CFG->dbhost	Replace the accountname with your Sql Azure account name
config.php (8)	$CFG->dbname	Make sure there is a “moodle19” database created in your Sql Azure account
config.php (9)	$CFG->dbuser	Replace username by the actual Sql Azure username. Replace accountname with your Sql Azure account name.
config.php (10)	$CFG->dbpass	Your Sql Azure account password
config.azure.php (7)	$CGF->wwwroot	Replace sitename with the web role account name.
config.azure.php (25)	$azureAccountName	This is your storage services account name.
config.azure.php (26)	$azureAccountKey	This is your storage services account key. It is a base 64-encoided key.

6. Right click on the Unosquare.MoodleAzure Project and select “Publish…”. You can deploy the service directly through Visual Studio by selecting the “Deploy your Cloud Service to Windows Azure” option and setting up your deployment credentials. It might take around 15 minutes to upload the service package, and create the Azure Virtual Machine.
7. Open up a browser and go to http://(sitename).cloudapp.net where sitename is the account name of your web role. The site will take some time to generate a response. If the response takes longer than 35 seconds, it will show a 500 error. This is because there is currently no way of telling FastCGI on Azure to allow for longer execution times. If you get a 500 error, refresh the page. If you get a message that an upgrade is already running, click the (!!!) link. If you get a page indicating that tables were created successfully and there is no continue button, refresh the page. Repeat this procedure until the database setup is complete.

Implementation Details: What Changed

In this section we go over the implementation details of what was added and what was changed in both, Moodle and the PHP Azure SDK.

config.azure.php – The first thing that is noticeable is the addition of the config.azure.php file. In this file we detect whether the application is running in the cloud. The “RoleRoot” environment variable is set when this is the case. Configuration values for table and blob storage services are also provided. We also register the ‘azure://’ custom stream wrapper Finally, the table storage-based session handler from the PHP Azure SDK is registered.

adodb-sqlsrv.inc.php – This file was originally based on the adodb-mssql.inc.php file. It was however heavily modified to use the sqlsrv extension instead. One of the more notable differences is the use of

Sql Azure system tables for metadata extraction. Some of the queries to extract column names, indexes, object types, and the like, were querying metadata tables that simply do not exist in in Sql Azure. They do however have an equivalent. The other obvious change is the way queries are executed. Notice the use of a Forward-only cursor when performing query execution (line 649). The reason behind this is that under Keyset mode, the database needs to create a temporary table which works in Azure, but it does not yield an acceptable performance.

formslib.php – Only a couple of lines changed here (111 and 837). These changes ensure that the form actions do not point to the Azure load balancer port 20000. When deploying Azure applications, PHP will rewrite the host (HTTP_HOST environment variable) to be the same as the one being accessed, but on a different port – the load balancer's --. This is a known issue documented in: http://social.msdn.microsoft.com/Forums/en/windowsazure/thread/fea38c47-cd6e-4e6a-a49d-aeab0f0dc686

gdlib.php – Turns out that imagejpeg, imagepng, and imagegif of the GD2 library does not support saving images using a registered stream. The workaround was to create the Azure versions of these functions (starting on line 325): ImageJpegAzure, ImagePngAzure, and ImageGifAzure. These functions create a temporary file to write the image file using GD2's native functions, and then write the file contents to the specified blob. All the calls to imagejpeg, imagegif, and imagepng were replaced with their Azure equivalents (lines 244 and 313).

graphlib.php – The only change in this file was the importing of gdlib.php at the beginning of the file and the replacement of all the calls to imagejpeg, imagegif, and imagepng with their azure equivalents.

field.class.php – Lines 448 and 255 replace imagejpeg and imagepng with their Azure equivalents as well. The importing of the gdlib.php script is done in the top of the script.

Storage.php – The PHP Azure SDK implements 2 policies: RetryN and NoRetry. The default policy to use when connecting to storage services is the NoRetry policy. After some testing, we decided to change the default policy to RetryN because we discovered that sometimes calls to storage services do fail. See line 201.

Stream.php – This is the file that contains the most modifications and additions. For instance, this implementation keeps a cache of blob instances and container names. This makes the stream wrapper more efficient because it does not have to fetch blob metadata or check whether a container exists every time an operation is required. The caches are only kept per call, and if within the same call a new blob is created, the caches get invalidated through the invalidateCache function. The caches are only valid for the current request. Whenever there is a file added, or the simulated file system changes, the cache is also invalidated. The getFileName function also changed. Mostly because we need a way to simulate multi-level folder structures in blobs. This is done by replacing the forward slash '/' with 3 underscores '___' . In this way, whenever the PHP application needs to list "files" within a "folder" we will just take the blobs that start with the given folder prefix. For example, if file 'azure://users/1/picture/1.jpg' is requested, we will need to read the blob named 'moodledata/users___1___picture___1.jpg'.

Opening a stream for appending (see line 256) would throw an exception whenever the blob did not exist before. But the append mode of a PHP stream is supposed to attempt to create a file. Therefore, we will only try to read the blob. If it doesn't exist, we will already have the temp file we are writing to and when closing the stream the temp file will be automatically written to the blob with the same name. Another challenge was to correctly handle the usrl_stat method call. Whenever the operating system needs to know the metadata of a files system entry (files or folders), it needs to fill out such information in the url_stat method of the stream wrapper. But in the original implementation, there was too little metadata available to simulate folders. At the OS level, the differentiation between files and folders is stored in the 'mode' entry. Folder modes are represented by 040777 and file modes are represented by 010066 (see line 566). Thus, blobs stored with a 'simulated-folder' content encoding will return a 040777 mode. Which takes us to the next method call: mkdir. Mikdir is the method call to create folders. As stated before, we are simulating folders with blobs. We store the folder-simulating blobs with a content encoding of 'simulated-folder'. See line 620 for further details. The rmdir method (see line 672) also needed modification because when deleting a folder, we need to delete all of its contents. In Azure, that means we need to delete all the blobs starting with the folder prefix. Finally, the dir_opendir implementation was changed so it would allow for listing only blobs with the specified path prefix.

Horizontally Scaling the Application

If you are not too experienced with Azure, you might be asking yourself at this point: How do I make this application scale horizontally? Is the amount of effort required to migrate an application to the cloud, worth it? The answer is simple. Expand the Unosquare.MoodleAzure project in your solution. Open the ServiceConfiguration.cscfg file. In the fourth line, you should see an node that called "Instances". Let's say you want to handle twice the amount of users you were handling before. Simply change the "count" attribute to 4. The best part is that you do not have to redeploy the solution. You can go into the Windows Azure portal, select your deployment, and click on "configure". This will open up the service configuration file and change it directly in the deployed solution. Just remember that for Azure's SLA to take effect you will need to set the instance count to at least 2.

Conclusion

There were around 500 lines of code changed or added in this particular scenario. Relatively speaking, the amount of code changes is not too significant. What is significant is the amount of debugging and the required knowledge of Azure services required to complete the task. As a result, we end up with and application that is easily scalable and provides the ability to handle an increasingly growing user base.